View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Journal of Discrete Algorithms 10 (2012) 23–34

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Indeterminate string inference algorithms ☆,☆☆

Sumaiya Nazeen *, M. Sohel Rahman, Rezwana Reaz

AℓEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh

ARTICLE INFO

Article history:
Available online 11 August 2011

Keywords:
Indeterminate string inference
Border array
Suffix array
LCP array

ABSTRACT

Regularities in indeterminate strings have recently been a matter of interest because of their use in the fields of molecular biology, musical text analysis, cryptanalysis and so on. In this paper, we study the problem of reconstructing an indeterminate string from a border array. We present two efficient algorithms to reconstruct an indeterminate string from a valid border array – one using an unbounded alphabet and the other using minimum sized alphabet. We also propose an $O(n^2)$ algorithm for reconstructing an indeterminate string from suffix array and LCP array.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

An indeterminate string (also referred to as degenerate strings in the literature) is a generalization of a (regular) string in which each position contains either a single character or a nonempty set of characters. When each position contains a single character, it reduces to a regular string. In recent years, the study of indeterminate strings has drawn a lot of attention. Indeterminate strings are extensively used in molecular biology. They are specially used to model biological sequences (e.g., FASTA format for representing either nucleotide sequences or peptide sequences). In fact, they are very effective in expressing polymorphism in such sequences (e.g., Single Nucleotide Polymorphism (SNP), polymorphism of protein coding regions caused by redundancy of genetic code or polymorphism in binding site sequences of a family of genes). Regularities like covers in indeterminate strings can be used to find tandem repeats of DNA sequences which help to determine individual's hereditary behavioral traits [5]. Not only in molecular biology but also in cryptanalysis, musical text analysis and search engine techniques, indeterminate strings have important usage.

The problems of indeterminate pattern matching [15,18,19,22] and finding regularities in indeterminate strings [2,3,5,14,21] have been addressed with great enthusiasm over the last decade. However, despite several results on regular string inference in the literature [9,11,13], the problem of indeterminate string inference is yet to be explored.

Franěk et al. [13] first introduced the problem of string inference from border arrays. They presented an online linear time algorithm to verify a given border array for some string on an unbounded alphabet. Duval et al. [11] gave an online linear time algorithm for bounded alphabet to solve the same problem. Bannai et al. [4] solved the problem of regular string inference from directed acyclic subsequence graph and directed acyclic word graph, and from suffix array using minimal size alphabet in linear time. A linear time and space method was presented by Duval and Lefebvre [10] to determine whether two words have the same suffix permutations for a given ordered alphabet. This work led to a method for generating a Lyndon word randomly in linear time or for computing the set of Lyndon words of length n. Very recently, Tomohiro et al. [16] proposed a way to verify whether a given integer array is a valid parameterized border array (p-border array) for a

☆ The authors' names are in the alphabetical order of their last names.

☆☆ This research was part of the undergraduate research work of Nazeen and Reaz under the supervision of Rahman.

* Corresponding author.
E-mail addresses: nazeen@cse.buet.ac.bd (S. Nazeen), msrahman@cse.buet.ac.bd (M.S. Rahman), rimpi@cse.buet.ac.bd (R. Reaz).

1570-8667/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2011.08.002

binary alphabet. Also, they presented two linear time algorithms – one for computing all the binary parameterized strings sharing a given p-border array and one for computing all p-border arrays of length at most n for binary alphabet. They further extended their work in [17] by giving an $O(n^{1.5})$-time $O(n)$-space algorithm to verify if a given integer array of length n is a valid p-border array for an unbounded alphabet. Crochemore et al. [9] presented an efficient algorithm that can reconstruct (regular) strings from a given valid *cover array*. In this paper, we address the problem of reconstruction of indeterminate strings and try to devise some novel reconstruction algorithms. In particular, first we present an algorithm for reconstruction of indeterminate strings from input border array using an unbounded alphabet. Then we modify this algorithm to use least sized alphabet. Finally we present another algorithm for reconstruction of degenerate strings from suffix array and LCP array.

The rest of this paper is organized as follows. Section 2 presents some definitions and notations used throughout the paper. Section 3 discusses some important properties of border array, extends them for indeterminate strings and formally defines the problems handled in this paper. In Section 4 we describe the algorithms and main findings. Finally, Section 5 concludes the paper.

2. Preliminaries

A string x is a finite sequence of symbols drawn from an alphabet Σ, where $|\Sigma| = k$ and $\sigma[i]$ denotes the ith symbol of Σ.

Let λ_i, $|\lambda_i| \geqslant 1$, $1 \leqslant i \leqslant K$, $K > k$, be pairwise distinct subsets of the alphabet Σ. We define a new alphabet $\Sigma' = \{\lambda_1, \lambda_2, \ldots, \lambda_K\}$ and a new relation (*indeterminate*) *match* denoted by the operator "$\approx$" on Σ' as follows:

$$\text{for every } \lambda_i, \lambda_j \in \Sigma', \quad \lambda_i \approx \lambda_j \text{ if and only if } \lambda_i \cap \lambda_j \neq \emptyset.$$

In a string x on an alphabet Σ', a position i is said to be *indeterminate* if and only if $|x[i]| \geqslant 2$, and $x[i]$ itself is said to be an *indeterminate* letter. A string that may contain indeterminate letters is said to be *indeterminate*. Two indeterminate strings x and y are said to (*indeterminate*) *match* if and only if they are of the same length and the letters in corresponding positions match, i.e., $u[1..n] \approx v[1..n] \Leftrightarrow \forall i \in \{1, 2, \ldots, n\}, u[i] \approx v[i]$. However, the *indeterminate match* ($\approx$) relation is not transitive. That is, if $\mu \approx \lambda$ and $\lambda \approx \nu$, it does not necessarily imply that $\mu \approx \nu$. For example, for $x = [ab][bc][cd]$, $x[1] \approx x[2]$ and $x[2] \approx x[3]$ but $x[1] \not\approx x[3]$. This idea seems to have first been mentioned in [12] and appeared in various forms such as generalized strings [1], subset matching [7,8], partial words [6], degenerate strings [20], and indeterminate strings [21].

We recall that, a string w is a *factor* of string x if $x = uwv$ for two strings u and v. It is a *prefix* of x if u is empty and *suffix* of x if v is empty. A *border* u of a regular string $x = x[1..n]$ is a proper *prefix* of x that is also a *suffix* of x; thus $u = x[1..b] = x[n - b + 1..n]$ for some $b \in 0..n - 1$. The definition of *border* of *indeterminate string* is a natural extension of the usual border based on the relation *match* ($\approx$) as described in [14]: a border of an indeterminate string $x = x[1..n]$ on Σ' is a proper prefix $x[1..b]$ such that $x[1..b] \approx x[n - b + 1..n]$. It may be noted that in [14], two kinds of borders, namely, the quantum border and the deterministic border were defined for indeterminate strings. For the former, an indeterminate letter is allowed to match two or more distinct letters in a single matching process, whereas the latter restricts an indeterminate letter to match only one regular letter in a single matching process. We restrict ourselves to the study of quantum border.

The *prefix array* $\pi = \pi[1..n]$ of a regular string $x = x[1..n]$ is an integer array in which $\pi[1] = 0$ and, for $i > 1$, $\pi[i] = p$ if and only if p is the largest integer such that $x[i..i + p - 1] = x[1..p]$. The definition of prefix array is extended for indeterminate strings as follows: the *prefix array* $\pi = \pi[1..n]$ of an indeterminate string $x = x[1..n]$ is an integer array in which $\pi[1] = 0$, and for $i > 1$, $\pi[i] = p$ if and only if p is the largest integer such that $x[i..i + p - 1] \approx x[1..p]$. This data structure is very interesting since it describes all the borders of every prefix of a string irrespective of whether the string is regular or indeterminate [21].

However, the border array is the data structure that is most commonly used to encode the border information of strings. The *border array* of a regular string $x = x[1..n]$ is an integer array $\beta = \beta[1..n]$ such that, for every $i \in \{1, 2, \ldots, n\}$, $\beta[i]$ is the length of the longest border of $x[1..i]$. In fact, since every border of any border of x is also a border of x, β encodes all the borders of all the borders of every prefix of x. Due to the *nontransitivity* of indeterminate match ($\approx$), this convenient relation does not hold for borders of indeterminate strings i.e., a border of a border of an indeterminate string is not necessarily a border of that string [21]. For example, given $x = [a][ab][b]$, $x[1..2]$ has borders of length 1 and 0, while $x[1..3]$ has a border of length 2 but not of length 1. Hence, an integer array consisting of the lengths of the longest borders of an indeterminate string x at every position, cannot encode all the borders of every prefix of x. For this reason, we extend the definition of a border array of an indeterminate string as follows:

Definition 1. The **border array** $B = B[1..n]$ of an indeterminate string $x = x[1..n]$ on Σ' is an array such that, for every $i \in \{1, 2, \ldots, n\}$, $B[i]$ is the list of the lengths of all the borders of each prefix $x[1..i]$ sorted in descending order. In other words, $B[i]$ is a list of the form $B^1[i], B^2[i], \ldots, B^m[i]$ where $B^k[i]$ is the length of the k-th longest border of $x[1..i]$, for $1 \leqslant k \leqslant m$. Essentially, $B^m[i] = 0$, the length of empty border of $x[1..i]$. The 0 entry is not mentioned unless it is the only entry at any position.

We give an example for clarity in Table 1.

Table 1
Border Array of the indeterminate string $[a][abc][ab][ab][b][a][ab][c]$.

i	1	2	3	4	5	6	7	8
$x[i]$	a	a b c	a b	a b	b	a	a b	c
$B[i]$	0	1	2 1	3 2 1	4 3 2	4 3 1	5 4 2 1	2

Table 2
Suffix array Ψ and LCP array Π of the regular string $x = abcabaa\$$.

i	$\Psi[i]$	Lexicographically sorted suffixes of $x = abcabaa\$$	$\Pi[i]$
1	8	\$	0
2	7	$a\$$	0
3	6	$aa\$$	1
4	4	$abaa\$$	1
5	1	$abcabaa\$$	2
6	5	$baa\$$	0
7	2	$bcabaa\$$	1
8	3	$cabaa\$$	0

Now we briefly discuss the basics of suffix array and LCP (Longest Common Prefix) array. Let, Σ be an ordered alphabet and $x = x[1]x[2]\dots x[n-1] \in \Sigma^{n-1}$ be a string over Σ of length $n-1$. Let \$ be a character not contained in Σ, and assume $\$ < c$ for all $c \in \Sigma$. Following the convention in the literature, we consider a \$-padded extension of string x denoted by $x^+ = x\$$. For ease of illustration, we'll refer to x^+ by x. For $1 \leqslant i \leqslant n$, let $s_i(x) = x[i]x[i+1]\dots x[n]$ indicate the ith nonempty suffix of x; the starting position i is called the *suffix number*.

The **suffix array** Ψ of x is a permutation of the suffix numbers $\{1, 2, \dots, n\}$ according to the lexicographical ordering of n suffixes of x. More precisely, for all pairs of indices (k, l) of Ψ, $1 \leqslant k < l \leqslant n$, $s_{\Psi[k]}$ is lexicographically smaller than $s_{\Psi[l]}$. Given, two strings v and w, we write $\mathrm{LCP}(v, w)$ to denote the length of their *longest common prefix*. Thus we define **LCP array** to be an integer sequence $\Pi = \Pi[1]\Pi[2]\dots\Pi[n]$ such that,

- $\Pi[1] = 0$.
- $\Pi[i] = LCP(s_{\Psi[i]}, s_{\Psi[i-1]}), 2 \leqslant i \leqslant n$.

For regular strings these two definitions give rise to two interesting equality conditions:

Condition 1.

1. $x[\Psi[i]..\Psi[i] + \Pi[i] - 1] = x[\Psi[i-1]..\Psi[i-1] + \Pi[i] - 1]$.
2. $x[\Psi[i] + \Pi[i]] \neq x[\Psi[i-1] + \Pi[i]]$.

We give an example for clarity in Table 2.

We now extend the notions of suffix array and LCP array for indeterminate strings. By indeterminate string $x = x[1]x[2]\dots x[n]$, here, we denote the \$-padded extension of indeterminate string $x[1..n-1]$, i.e. for $1 \leqslant i < n$, $x[i] \in \Sigma'$ and $x[n] = \{\$\}$.

We define the *lexicographical ordering* of the letters of alphabet Σ' as follows:

Definition 2 (*Lexicographical ordering of the indeterminate alphabet* Σ'). For any $\lambda_i, \lambda_j \in \Sigma'$,

- If $\lambda_i \cap \lambda_j = \emptyset$, then if $\exists u \in \lambda_i$ is lexicographically smaller than $\forall v \in \lambda_j$ then, λ_i is lexicographically smaller than λ_j, else λ_i is lexicographically greater than λ_j.
- If $\lambda_i \cap \lambda_j \neq \emptyset$, then
  - if $\lambda_i - (\lambda_i \cap \lambda_j) = \emptyset$ and $\lambda_j - (\lambda_i \cap \lambda_j) \neq \emptyset$ then, if $\exists u \in \lambda_i$ lexicographically smaller than $\exists v \in \lambda_j - (\lambda_i \cap \lambda_j)$, then λ_i is lexicographically smaller than λ_j,
  - if $\lambda_i - (\lambda_i \cap \lambda_j) \neq \emptyset$ and $\lambda_j - (\lambda_i \cap \lambda_j) = \emptyset$ then, if $\exists u \in \lambda_j$ lexicographically smaller than $\exists v \in \lambda_i - (\lambda_i \cap \lambda_j)$, then λ_i is lexicographically greater than λ_j,
  - if $\lambda_i - (\lambda_i \cap \lambda_j) \neq \emptyset$ and $\lambda_j - (\lambda_i \cap \lambda_j) \neq \emptyset$ then, if $\exists u \in \lambda_i - (\lambda_i \cap \lambda_j)$ is lexicographically smaller than $\forall v \in \lambda_j - (\lambda_i \cap \lambda_j)$ then, λ_i is lexicographically smaller than λ_j. Otherwise, λ_i is lexicographically greater than λ_j.

Table 3
Lexicographically ordered Σ and Σ'.

$\Sigma =$	$\{a, c, g, t\}$
$\Sigma' =$	$\{[a], [ac], [acg], [acgt], [act], [ag], [agt], [at], [c], [cg], [cgt], [ct], [g], [gt], [t]\}$

Table 4
Suffix array Ψ and LCP array Π of the indeterminate string $x = [at][c][cg][at][c][acgt][\$]$.

i	$\Psi[i]$	Lexicographically sorted suffixes of $x = [at][c][cg][at][c][acgt][\$]$	$\Pi[i]$
1	7	$[\$]$	0
2	6	$[acgt][\$]$	0
3	4	$[at][c][acgt][\$]$	1
4	1	$[at][c][cg][at][c][acgt][\$]$	3
5	5	$[c][acgt][\$]$	0
6	2	$[c][cg][at][c][acgt][\$]$	2
7	3	$[cg][at][c][acgt][\$]$	1

Also Condition 1 (*equality condition*) changes as follows based on indeterminate match ($\approx$) relation:

Condition 2.

1. $x[\Psi[i]..\Psi[i] + \Pi[i] - 1] \approx x[\Psi[i-1]..\Psi[i-1] + \Pi[i] - 1]$.
2. $x[\Psi[i] + \Pi[i]] \not\approx x[\Psi[i-1] + \Pi[i]]$.

We give illustrative examples of lexicographically ordered indeterminate alphabet Σ' and the suffix array and LCP array of an indeterminate string in Tables 3 and 4 to clarify the above ideas.

3. Validation of border arrays

Duval et al. [11] used two important conditions for checking the validity of a border array of a regular string. The necessity and sufficiency of those conditions were proved in [13]. We approach the validation of border array of an indeterminate string in a similar way.

Suppose that, we have a valid border array until position $i-1$. For $i \geqslant 2$, we say an integer $j+1$ is a *candidate-length* (i.e., "candidate" to be the length) of a border of $x[1..i]$, if $j \in B[i-1]$, i.e., $x[1..j]$ is a border of $x[1..i-1]$. Thus, the decreasing sequence of candidate-lengths of borders of $x[1..i]$ is

$$\pi_i = \left\{1 + B^1[i-1], 1 + B^2[i-1], \ldots, 1 + B^m[i-1]\right\}$$

where $B^m[i-1] = 0$.

We say that an array $B[1..n]$ is a *valid border array* if and only if it is the border array of at least one indeterminate string x with n positions (i.e., having length n).

Clearly, the only border of $x[1]$ is necessarily an empty word. Thus we must have $B[1] = \{0\}$, irrespective of any strings. Also, as has been discussed above, the list of lengths of the nonempty borders of $x[1..i]$, $B[i]$ is taken from π_i. Thus we have a necessary condition for an array $B[1..n]$ to be valid as follows:

Condition 3.

1. $B[1] = \{0\}$,
2. $B[i] \subseteq \{0\} \cup \pi_i$, for $2 \leqslant i \leqslant n$,
   a. If $x[1..i]$ has the empty word for its only border then we have $B[i] = \{0\}$.
   b. If $x[1..i]$ has nonempty borders then $B[i] = \{j+1 | j+1 \in \pi_i \text{ and } x[i] \approx x[j+1]\}$.

Theorem 1. *For every $n \geqslant 1$, an array of n lists of integers, $B[1..n]$ is a border array of some indeterminate string $x[1..n]$ if and only if Condition 3 is satisfied.*

Proof. To prove sufficiency, let, $B[1..n]$ be an array of n lists of integers such that Condition 3 is satisfied. We show by induction that, $B[1..n]$ is the border array of some indeterminate string $x[1..n]$.

Basis: Clearly, the only border of any unit length indeterminate string $x[1]$, is necessarily an empty word. Thus, $B[1] = \{0\}$ is the border array of any unit length indeterminate string. So, the result holds for $n = 1$.

Let, for $n \geqslant 2$ and some $i \in 2..n$, $B[1..i-1]$ satisfies Condition 3 and is the border array of some indeterminate string $x[1..i-1]$. We show that, $B[1..i]$ must be a border array of the indeterminate string $x[1..i]$.

Induction: There are two cases.

Case 1 $B[i] = \{0\}$. In this case $B[1..i]$ is a border array of an indeterminate string $x[1..i-1]\mu$, where the letter μ is chosen so that, it does not extend any border of $x[1..i-1]$. That is, choice of μ ensures that, $x[1..i]$ has only an empty word as for its border.

Case 2 $B[i] \subseteq \pi_i$. We consider some $\nu \in B[i]$ such that, $\nu = B^p[i-1]+1$ for some integer $p \geqslant 1$. By induction hypothesis, $B[1..i-1]$ is the border array of some indeterminate string $x[1..i-1]$. Hence, ν is a candidate length of a border of some indeterminate string $x[1..i-1]\mu$, where the letter μ is chosen so that, $\mu \approx x[\nu]$. For every nonzero ν in $B[i]$ such μ can be chosen and collection of all these choices will give us the desired $x[i]$. Thus we can find at least one indeterminate string $x[1..i]$, with $x[i]$ at its ith position, for the array $B[1..i]$.

Thus, we can conclude that $B[1..n]$ is the border array of some indeterminate string $x[1..n]$ given, $B[1..n]$ satisfies Condition 3.

To prove necessity we assume that, $B[1..n]$ is a border array of some indeterminate string $x[1..n]$, $n \geqslant 1$, and we prove that, $B[1..n]$ satisfies Condition 3.

For $n = 1$, any string $x[1]$ has only a border of length zero and this is listed into the border array by making $B[1] = \{0\}$. Thus, Condition 3.1 is satisfied.

We may suppose, $n \geqslant 2$ and for $2 \leqslant i \leqslant n$, $B[1..i]$ is a border array of $x[1..i]$. Now we prove Condition 3.2 for $n \geqslant 2$. There are two cases.

Case 1: Let, $x[1..i]$ have a nonempty border of length j. Since, $B[1..i]$ is the border array of $x[1..i]$, $j \in B[i]$ and $x[i] \approx x[j]$. For j to be in $B[i]$, it must be in π_i also, since π_i lists all the candidate lengths of borders at position i of x. We suppose however that $j \notin \pi_i$ and derive a contradiction.

Any nonzero border of $x[1..i]$ must be obtained by extending any border of $x[1..i-1]$ (including zero border). So, to have a nonzero border of length j of $x[1..i]$, $x[1..i-1]$ must have a border of length $j-1$. In other words, the value $j-1$ must be in $B[i-1]$. Hence, j must be in π_i. Thus, contradiction arises. So, Condition 3.2 is satisfied.

Case 2: Let, $x[1..i]$ have only a border of length zero. Since $B[1..i]$ is the border array of $x[1..i]$, $B[i]$ must have a single entry of zero at the ith position. Thus, Condition 3.2 is satisfied. □

3.1. Problem definition & important properties

Below we formally define the problems handled in this paper.

Problem 1. Given a valid border array B of length n, reconstruct an indeterminate string of length n on an unbounded alphabet.

Problem 2. Given a valid border array B of length n, reconstruct an indeterminate string of length n on a minimum-sized alphabet.

Problem 3. Given a suffix array Ψ and an LCP array Π, each of length n, reconstruct an indeterminate string of length n on an unbounded alphabet.

The following properties of border arrays of indeterminate strings are noteworthy:

Property 1. *(See [14].) Since border array $B[1..n]$ contains list of the lengths of all borders of each prefix of a string $x[1..n]$, we have $B^p[i] = B^q[i-1]+1$ when $x[B^q[i-1]+1] \approx x[i]$, or $B^p[i] = 1$ when $x[1] \approx x[i]$ or else $B^p[i] = 0$; where, $1 \leqslant i \leqslant n$, $1 \leqslant p \leqslant |B[i]|$ and $1 \leqslant q \leqslant |B[i-1]|$.*

Property 2. *(See [5].) The number of borders of an indeterminate string is bounded by a constant on average.*

4. New algorithms

4.1. BrAyISRUn algorithm

We first propose an algorithm for Problem 1. We call this algorithm the BrAyISRUn (**B**order **A**rray **I**ndeterminate **S**tring **R**econstruction from **Un**bounded **A**lphabet) algorithm. Given an array $B[1..n]$, BrAyISRUn determines whether $B[1..n]$ is a valid border array for at least one indeterminate string and if so then, it constructs one such indeterminate string. Before presenting the algorithm, we first need to present some relevant definitions and notions.

```
BrAyISRUn(B, n)
 1  if B[1] ≠ {0}
 2    then return B invalid at index 1
 3  x[1] ← {σ[1]}
 4  k[1] ← 1
 5  for i ← 2 to n
 6  do
 7     π ← {B^j[i − 1] + 1 | 1 ⩽ j ⩽ |B[i − 1]|} ∪ {1}
 8     x[i] ← ∅
 9     A′ ← (⋃_{j∈π−B[i]} x[j])
10     k[i] ← k[i − 1]
11     A ← ∅
12     for j ← 1 to |B[i]|
13     do
14        if B^j[i] ≠ 0
15          then
16                if B^j[i] ∉ π
17                  then return B invalid at index i
18                A ← x[B^j[i]] − A′
19                if A ≠ ∅
20                  then x[i] ← x[i] ∪ A
21                  else
22                        k[i] ← k[i − 1] + 1
23                        A ← {σ[k[i]]}
24                        x[i] ← x[i] ∪ A
25                        x[B^j[i]] ← x[B^j[i]] ∪ A
26
27          else
28                k[i] ← k[i] + 1
29                A ← {σ[k[i]]}
30                x[i] ← x[i] ∪ A
31
32  return x
```

Fig. 1. Algorithm BrAyISRUn.

Given an indeterminate string x of length n, we define $\Sigma_i \subseteq \Sigma$ to be the set of symbols used by the (sub)string $x[1..i]$. Suppose we are reconstructing an indeterminate string $x = x[1..n]$ from a border array $B[1..n]$. Assume that, we have successfully reconstructed $x[1..i-1]$. We use ψ_i to denote the new set of characters introduced in $x[i]$, i.e., $\psi_i = \Sigma_i - \Sigma_{i-1}$. Now, we want to extend $x[1..i-1]$ to get $x[1..i]$ based on $B[1..i]$. We denote by A'_i the set of symbols that are not allowed at position i, i.e., $A'_i = \bigcup_{j\in\pi_i - B[i]} x[j]$. On the other hand, we denote by A_i the set of symbols that can be assigned to $x[i]$. We now have the following lemma.

Lemma 1. *For every indeterminate string $x[1..i]$ we have*

1. *If $B^p[i] \neq 0$ for $1 \leqslant p \leqslant |B[i]|$ then*

$$x[i] \approx x\big[B^p[i]\big],\ B^p[i] \in \pi_i \quad \text{and} \quad A_i = \psi_i \cup \Bigg(\bigcup_{1\leqslant p\leqslant |B[i]|} \big(x\big[B^p[i]\big] - A'_i\big)\Bigg).$$

2. *If $B^p[i] = 0$ is the only entry of $B[i]$, then $B^p[i] \notin \pi_i$ and $A_i = \psi_i$.*

Proof. 1. If $B^p[i] \neq 0$ for $1 \leqslant p \leqslant |B[i]|$ then $B^p[i]$ is a candidate in π_i and as $B^p[i] \in B[i]$ so $x[i]$ must match $x[B^p[i]]$. But, valid candidate set A_i cannot contain any symbol from the set A'_i, because, otherwise, validity of B will be violated. So, A_i includes $x[B^p[i]] - A'_i$ for each p. When $x[B^p[i]] - A'_i = \emptyset$, A_i needs to incorporate new characters. The set of new characters is denoted by ψ_i. Thus, $A_i = \psi_i \cup (\bigcup_{1\leqslant p\leqslant |B[i]|}(x[B^p[i]] - A'_i))$.

2. $B^p[i] = 0$ implies that, $x[1..i]$ has an empty word border and there is no candidate $j+1 \in \pi_i$ such that $x[i] \approx x[j+1]$. Thus valid candidate set $A_i = \psi_i$. □

The steps of BrAyISRUn are formally presented in Fig. 1. We assume that, we have an array σ representing an unbounded alphabet from which we take the basic letters. The BrAyISRUn algorithm takes an array $B[1..n]$ as input. Initially, it checks the necessary condition (Condition 3) for B to be valid. That is, it checks the trivial validity condition whether $B[1] = \{0\}$ or not (Condition 3.1). Then, for every position $2 \leqslant i \leqslant n$, it checks whether $B^p[i] \in \pi_i$, $1 \leqslant p \leqslant |B[i]|$, to ensure the validity of B (Condition 3.2). Algorithm BrAyISRUn terminates as soon as it finds a violation of the condition checked above. As long as the results of the above checking are positive, Algorithm BrAyISRUn constructs A'_i and A_i for each position $2 \leqslant i \leqslant n$ and calculates the alphabet size $k[i] = |\Sigma_i|$ to construct each prefix $x[1..i]$. Whenever some $B^p[i] \neq 0$, algorithm BrAyISRUn puts a character $v \in A_i$ into $x[i]$; v is also included in $x[B^p[i]]$ to maintain the validity of B. So, we have the following theorem.

	i	1	2	3	4	5	6	7	8	9	10
	$B[i]$	0	1	2	3 1	4 2	5 3 1	6 4 2 1	2 1	3 1	0

Itn.	i	1	2	3	4	5	6	7	8	9	10	$k[i]$	Explanation
0	$x[i]$	a										$k[1]=1$	
1	$x[i]$	a	a									$k[2]=1$	$\pi_2=\{1\}$ $A'_2=\emptyset$, $A_2=\{a\}$
2	$x[i]$	a	a b	b								$k[3]=2$	$\pi_3=\{2,1\}$, $A'_3=\{a\}$ for $B^1[3]=2$, new symbol 'b' $A_3=\psi_3=\{b\}$
3	$x[i]$	a	a b	b	a b							$k[4]=2$	$\pi_4=\{3,1\}$ $A'_4=\emptyset$, $A_4=\{a,b\}$
4	$x[i]$	a	a b	b	a b	b						$k[5]=2$	$\pi_5=\{4,2,1\}$ $A'_5=\{a\}$, $A_5=\{b\}$
5	$x[i]$	a	a b	b	a b	b	a b					$k[6]=2$	$\pi_6=\{5,3,1\}$ $A'_6=\emptyset$, $A_6=\{a,b\}$
6	$x[i]$	a	a b	b	a b	b	a b	a b				$k[7]=2$	$\pi_7=\{6,4,2,1\}$ $A'_7=\emptyset$, $A_7=\{a,b\}$
7	$x[i]$	a d	a b c	b	a b	b	a b	a b	c d			$k[8]=4$	$\pi_8=\{7,5,3,2,1\}$, $A'_8=\{a,b\}$ for $B^1[8]=2$, new symbol 'c' for $B^2[8]=1$, new symbol 'd' $A_8=\psi_8=\{c,d\}$ $A_2=A_2\cup\{c\}$, $A_1=A_1\cup\{d\}$
8	$x[i]$	a d	a b c	b e	a b	b	a b	a b	c d	e d		$k[9]=5$	$\pi_9=\{3,2,1\}$, $A'_9=\{a,b,c\}$ for $B^1[9]=3$, new symbol 'e' for $B^2[9]=1$, $x[1]-A'_9=\{d\}$ $A_9=\psi_9\cup(x[1]-A'_9)=\{e,d\}$ $A_3=A_3\cup\{e\}$
9	$x[i]$	a d	a b c	b e	a b	b	a b	a b	c d	e d	f	$k[10]=6$	$\pi_{10}=\{4,2,1\}$, $A'_{10}=\{a,b,c,d\}$ for $B^1[10]=0$, new symbol 'f' $A_{10}=\psi_{10}=\{f\}$

Fig. 2. Example run of algorithm BrAyISRUn.

Theorem 2. *Given a border array $B[1..n]$, the algorithm* BrAyISRUn *checks for its validity at every position and as long as it is valid it reconstructs an indeterminate string $x[1..n]$ on an unbounded alphabet for which $B[1..n]$ is a border array.*

The complexity of algorithm BrAyISRUn is analyzed below.

Theorem 3. *Algorithm* BrAyISRUn *runs in $O(N|\Sigma|)$ time, where N is the size of the border array B.*

Proof. The input of algorithm BrAyISRUn is a border array $B[1..n]$ with n positions. Note that each $B[i]$ is a list of integers. Assume that, m is the maximum number of elements in $B[i]$, $1\leqslant i\leqslant n$, i.e., $m=\max|B[i]|$, $1\leqslant i\leqslant n$. Then, the size N of input is $O(nm)$. Now, clearly $|\pi_i|$ and $|B[i]|$ are $O(m)$ and $|x[i]|$ can be at most $|\Sigma|$. For each position i, construction of π_i takes $O(m)$ time. Since $|\pi_i - B[i]|$ is also $O(m)$, construction of A'_i also requires $O(m|\Sigma|)$ time. Similarly, A_i can also be constructed in $O(m|\Sigma|)$ time. Thus, for n positions, the total running time of the algorithm is $O(nm|\Sigma|)$, i.e., $O(N|\Sigma|)$. □

Corollary 1. *Algorithm* BrAyISRUn *runs in linear time on average.*

Proof. Recall that, according to Property 2, the expected number of borders of an indeterminate string is bounded by a constant. Also, note that in the worst case $|\Sigma|$ asymptotically cannot exceed the total number of entries in the border array and on average it is bounded by a constant. Therefore, on average, the running time is $O(n)$, i.e., linear in the number of input size. □

Fig. 2 shows an example run of the algorithm.

```
BrAyISRin(B, n)
 1  if B[1] ≠ {0}
 2    then return B invalid at index 1
 3  x[1] ← {σ[1]}
 4  k[1] ← 1
 5  for i ← 2 to n
 6  do
 7     π ← {B^j[i − 1] + 1|1 ⩽ j ⩽ |B[i − 1]|} ∪ {1}
 8     x[i] ← ∅
 9     A′ ← (⋃_{j∈π−B[i]} x[j])
10     k[i] ← k[i − 1]
11     A ← ∅
12     for j ← 1 to |B[i]|
13     do
14        if B^j[i] ≠ 0
15          then
16                if B^j[i] ∉ π
17                  then return B invalid at index i
18                A ← x[B^j[i]] − A′
19                if A ≠ ∅
20                  then x[i] ← x[i] ∪ A
21                  else
22                        if j = 1
23                          then k[i] ← k[i − 1] + 1
24                                A ← {σ[k[i]]}
25                          else
26                                for m ← 1 to j − 1
27                                do
28                                   if B^j[i] ∈ B[B^m[i]]
29                                     then A ← A ∪ (x[B^m[i]] − A′)
30                                          break
31
32                                if m = j
33                                  then k[i] ← k[i] + 1
34                                       A ← {σ[k[i]]}
35                        x[i] ← x[i] ∪ A
36                        x[B^j[i]] ← x[B^j[i]] ∪ A
37          else
38                A ← σ[1..k[i]] − A′
39                if A = ∅
40                  then
41                        k[i] ← k[i] + 1
42                        A ← {σ[k[i]]}
43                x[i] ← x[i] ∪ A
44  return x
```

Fig. 3. Algorithm BrAyISRin.

4.2. BrAyISRin *algorithm*

Now we present a modified version of algorithm BrAyISRUn which reconstructs an indeterminate string using a minimum sized alphabet. We call this algorithm BrAyISRin (**B**order **A**rray **I**ndeterminate **S**tring **R**econstruction from M**in**imal Alphabet) algorithm. Before presenting algorithm BrAyISRin, we introduce some notation and lemmas.

As before, suppose we are reconstructing an indeterminate string $x = x[1..n]$ from a border array $B[1..n]$ and assume that we have successfully reconstructed $x[1..i-1]$. Now, we want to extend $x[1..i-1]$ to get $x[1..i]$ based on $B[1..i]$. Recall from Section 4.1 that, we use A'_i and A_i to denote the set of symbols that, respectively, are not allowed and allowed to be assigned to $x[i]$. Now we present an extended version of Lemma 1.2 below.

Lemma 2. *For every indeterminate string $x[1..i]$, if $B^p[i] = 0$ is the only entry in $B[i]$, then $B^p[i] \notin \pi_i$ and $A_i = \psi_i \cup (\Sigma_{i-1} - A'_i)$.*

Proof. $B^p[i] = 0$ implies that, $x[1..i]$ has an empty word as its border and there is no candidate $j+1 \in \pi_i$ such that $x[i] \approx x[j+1]$. Then, any $\nu \in \Sigma_{i-1} - A'_i$ can satisfy empty border at this position. If no such character is found then a new character is considered as a candidate. Thus valid candidate set $A_i = \psi_i \cup (\Sigma_{i-1} - A'_i)$. □

The algorithm BrAyISRin is formally presented in Fig. 3. BrAyISRin algorithm works exactly like the algorithm BrAyISRUn except for that it computes A_i slightly differently. In particular, it computes A_i following Lemmas 1.1 and 2 (instead of only Lemma 1.2). Note that, to reconstruct an indeterminate string with a minimum sized alphabet, the algorithm BrAyISRin keeps track of the size of the alphabet in the array k. We now have the following lemmas.

Lemma 3. *Let $x[1..n]$ be an indeterminate string and $k[1..n]$ be the array computed by the algorithm* BrAyISRin *given a valid border array B. Then, for $1 \leqslant i \leqslant n$ we have $k[i] = |\Sigma_{i-1} \cup A_i| = k[i-1] + |A_i| - |\Sigma_{i-1} \cap A_i|$.*

Proof. The proof immediately follows from the algorithm BrAyISRin and Lemma 2. □

Lemma 4. *Suppose given a valid border array $B[1..n]$, the algorithm* BrAyISRin *returns an indeterminate string $x[1..n]$ and computes the array $k[1..n]$. Then, the minimum cardinality of an alphabet required to build each prefix $x[1..i]$ is equal to $k[i]$.*

Proof. At any position i, the set added to x is A_i which may contain some old characters as well as some new characters as introduced per requirement according to Lemma 2. Thus at position i the minimal alphabet used is $\Sigma_{i-1} \cup A_i$. Hence, by Lemma 3, the result follows. □

The above discussion can be summarized in the following theorem.

Theorem 4. *Given a border array $B[1..n]$, the algorithm* BrAyISRin *checks for its validity at every position and as long as it is valid it reconstructs an indeterminate string $x[1..n]$ on a minimum sized alphabet for which $B[1..n]$ is a border array.*

The runtime analysis of algorithm BrAyISRin follows readily from the analysis of algorithm BrAyISRUn. The only extra work the former does is the calculation of $\Sigma_{i-1} - A'_i$ which can be done in $O(m|\Sigma|)$ time. Therefore we have the following results.

Theorem 5. *Algorithm* BrAyISRin *runs in $O(N|\Sigma|)$ time, where N is the size of the border array B.*

Corollary 2. *Algorithm* BrAyISRin *runs in linear time on average.*

Fig. 4 shows an example run of BrAyISRin algorithm.

4.3. SLISR *algorithm*

We now present a novel algorithm for inferring indeterminate strings from a given suffix array and an LCP array as a solution to Problem 3. In particular, given a suffix array, Ψ and an LCP array, Π, we propose an algorithm that builds an indeterminate string on an unbounded alphabet. We call this algorithm the SLISR algorithm(**S**uffix array and **L**CP array **I**ndeterminate **S**tring **R**econstruction).

The heart of our algorithm is the construction of *suffix graph* and assignment of characters based on that. We define the *suffix graph* as follows:

Definition 3. Given a suffix array, Ψ and an LCP array, Π, for any \$-padded string $x[1..n]$, a *suffix graph* $G = (V, E)$ is an undirected graph such that,

- $V = \{1, 2, \ldots, n-1\}$.
- $E = T \cup L$ where
  - $T = \{(\Psi[i] + k, \Psi[i-1] + k) | 0 \leqslant k < \Pi[i]\}$.
  - $L = \{(i, i) \mid i \text{ is an isolated position}\}$.

Fig. 5 shows a *Suffix Graph* constructed from given suffix array Ψ and LCP array Π. The algorithm SLISR is given in Fig. 7. The procedure BSG in Fig. 6, is used by algorithm SLISR to build the suffix graph. The steps of the algorithm are summarized below.

Step 1 (*Construction of suffix graph*): Using the given suffix array, Ψ and the LCP array, Π, a *suffix graph* $G = (V, E)$ is constructed by BSG.
Step 2 (*Character assignment*): For each edge of E, a new character is introduced, and assigned to both the endpoints.

The algorithm uses the following arrays:

- $\Psi[1..n]$: input suffix array.
- $\Pi[1..n]$: input LCP array.
- $\sigma[1..n]$: array of characters belonging to the alphabet Σ.
- $x[1..n]$: string inferred by the algorithm.

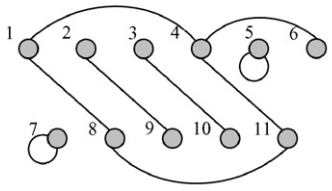
	i	1	2	3	4	5	6	7	8	9	10
	$B[i]$	0	1	2	3 1	4 2	5 3 1	6 4 2 1	2 1	3 1	0

Itn.	i	1	2	3	4	5	6	7	8	9	10	$k[i]$	Explanation
0	$x[i]$	a										$k[1]=1$	
1	$x[i]$	a	a									$k[2]=1$	$\pi_2=\{1\}$ $A'_2=\emptyset$, $A_2=\{a\}$
2	$x[i]$	a	a b	b								$k[3]=2$	$\pi_3=\{2,1\}$, $A'_3=\{a\}$ for $B^1[3]=2$, new symbol 'b' $A_3=\psi_3=\{b\}$
3	$x[i]$	a	a b	b	a b							$k[4]=2$	$\pi_4=\{3,1\}$ $A'_4=\emptyset$, $A_4=\{a,b\}$
4	$x[i]$	a	a b	b	a b	b						$k[5]=2$	$\pi_5=\{4,2,1\}$ $A'_5=\{a\}$, $A_5=\{b\}$
5	$x[i]$	a	a b	b	a b	b	a b					$k[6]=2$	$\pi_6=\{5,3,1\}$ $A'_6=\emptyset$, $A_6=\{a,b\}$
6	$x[i]$	a	a b	b	a b	b	a b	a b				$k[7]=2$	$\pi_7=\{6,4,2,1\}$ $A'_7=\emptyset$, $A_7=\{a,b\}$
7	$x[i]$	a c	a b c	b	a b	b	a b	a b	c			$k[8]=4$	$\pi_8=\{7,5,3,2,1\}$, $A'_8=\{a,b\}$ for $B^1[8]=2$, new symbol 'c' for $B^2[8]=1$, $B^2[8]\in B[B^1[8]]$ $A_8=\psi_8=\{c\}$ $A_2=A_2\cup\{c\}$, $A_1=A_1\cup\{c\}$
8	$x[i]$	a c e	a b c	b d	a b	b	a b	a b	c	d e		$k[9]=5$	$\pi_9=\{3,2,1\}$, $A'_9=\{a,b,c\}$ for $B^1[9]=3$, new symbol 'd' for $B^2[9]=1$, new symbol 'e' $A_9=\psi_9=\{d,e\}$ $A_3=A_3\cup\{d\}$, $A_1=A_1\cup\{e\}$
9	$x[i]$	a c e	a b c	b d	a b	b	a b	a b	c	d e	d	$k[10]=5$	$\pi_{10}=\{4,2,1\}$, $A'_{10}=\{a,b,c,e\}$ $A_{10}=\{a,b,c,d,e\}-A'_{10}=\{d\}$

Fig. 4. Example run of algorithm BRAYISRIN.

i	1	2	3	4	5	6	7	8	9	10	11	12
$\Psi[i]$	12	11	8	1	4	6	9	2	10	3	5	7
$\Pi[i]$	0	0	1	4	1	1	0	3	0	2	0	0

(a)



(b)

Fig. 5. (a) Input suffix array Ψ and LCP array Π, and (b) corresponding *Suffix Graph*.

Fig. 8 shows an example run of algorithm SLISR.

We now state and prove the results that are immediate from the algorithm.

Theorem 6. *When applied to a suffix array $\Psi[1..n]$ and LCP array $\Pi[1..n]$, the algorithm* SLISR *builds a* \$*-padded indeterminate string $x[1..n]$ on an unbounded alphabet satisfying both Ψ and Π.*

```
BSG(Ψ, Π, n)
 1  V ← {1, 2, ..., n}
 2  E ← φ
 3  for i ← 2 to n
 4  do for m ← 0 to Π[i] − 1
 5     do E ← E ∪ {edge(Ψ[i] + m, Ψ[i − 1] + m)}
 6        isConnected[Ψ[i] + m] ← isConnected[Ψ[i − 1] + m] ← 1
 7  for i ← 1 to n
 8  do if isConnected[i] = 0
 9        then E ← E ∪ {edge(i, i)}
10
11  return G(V, E)
```

Fig. 6. Procedure BSG.

```
SLISR(Ψ, Π, n)
 1  x[n] ← $
 2  G(V, E) ← BSG(Ψ, Π, n)
 3  k ← 0
 4  for each edge(i, j) in E
 5  do k ← k + 1
 6     x[i].add(σ[k])
 7     x[j].add(σ[k])
 8  return x
```

Fig. 7. Algorithm SLISR.

i	1	2	3	4	5	6	7	8
$\Psi[i]$	8	7	6	4	1	5	2	3
$\Pi[i]$	0	0	1	1	2	0	1	0

(a)

$E = \{(6, 7), (4, 6), (1, 4), (2, 5), (3, 3)\}$

(b)

i	1	2	3	4	5	6	7	8
$x[i]$	c	d	e	b c	d	a b	a	\$

(c)

Fig. 8. (a) Input suffix array & LCP array, (b) edge list of corresponding suffix graph, and (c) string inferred by algorithm SLISR.

Proof. The proof is clear from the intuitive construction of indeterminate string by the algorithm SLISR. The algorithm first constructs a *suffix graph* from the input suffix array Ψ and LCP array Π. This suffix graph embodies the matching conditions (Condition 2), stated in Section 2, in its edge relations such that each pair of matching positions are connected by an edge. Thus to match two connected positions the algorithm gives each end point of an edge same letters. Finally isolated positions receive characters for the self-loops defined for them in suffix graph. Since a position can be involved in multiple edges so it can receive multiple characters. Thus the string constructed by the algorithm may have set of characters at positions and hence is an indeterminate string. □

Theorem 7. *The running time of the algorithm* SLISR *is* $O(n^2)$ *in the size of input.*

Proof. The running time of algorithm SLISR depends essentially on the construction of suffix graph $G = (V, E)$, especially the edge set E. The main loop of the algorithm runs at most $|E|$ times. So the running time is $O(|E|)$. In the worst case, suffix graph can have $O(n^2)$ edges. Then the running time becomes $O(n^2)$. □

5. Conclusion

In this paper, we have presented efficient algorithms for verifying a border array of some indeterminate strings. In the case where the input is a valid border array, we have been able to efficiently infer an indeterminate string on both an unbounded alphabet and a least sized alphabet satisfying the border array. We have also presented an algorithm for reconstructing an indeterminate string from a given suffix array and an LCP array on unbounded alphabet. To the best of our knowledge this is the first attempt to solve these problems. Future research may be carried out for improving the running time of the given algorithms using advanced data structures (e.g., self-balanced tree), and for devising similar reconstruction algorithms for indeterminate strings considering other data structures (e.g., cover array).

Acknowledgements

The authors would like to acknowledge the fruitful comments of the anonymous reviewers which helped to improve the presentation of the paper a lot.

References

[1] K.R. Abrahamson, Generalized string matching, SIAM J. Comput. 16 (1987) 1039–1051.
[2] P. Antoniou, M. Crochemore, C.S. Ilipoulos, I. Jayasekera, G.M. Landau, Conservative string covering of indeterminate strings, in: Proceedings of the Prague Stringology Conference, 2008, pp. 108–115.
[3] P. Antoniou, C.S. Iliopoulos, I. Jayasekera, W. Rytter, Computing repetitive structures in indeterminate strings, in: Proceedings of the 3rd IAPR International Conference on Pattern Recognition in Bioinformatics, 2008, pp. 108–115.
[4] H. Bannai, S. Inenaga, A. Shinohara, M. Takeda, Inferring strings from graphs and arrays, in: Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science, in: LNCS, vol. 2747, Springer, Berlin/Heidelberg, 2003, pp. 208–217.
[5] M.F. Bari, M.S. Rahman, R. Shahriar, Finding all covers of an indeterminate string in $O(n)$ time on average, in: Proceedings of the Prague Stringology Conference, Czech Technical University in Prague, Czech Republic, 2009, pp. 263–271.
[6] F. Blanchet-Sadri, R.A. Hegstrom, Partial words and a theorem of Fine and Wilf revisited, Theoret. Comput. Sci. 270 (2002) 401–419.
[7] R. Cole, R. Hariharan, Tree pattern matching to subset matching in linear time, SIAM J. Comput. 32 (2003) 1056–1066.
[8] R. Cole, R. Hariharan, P. Indyk, Tree pattern matching and subset matching in deterministic $O(n \log^3 n)$-time, in: SODA, 1999, pp. 245–254.
[9] M. Crochemore, C.S. Iliopoulos, S.P. Pissis, G. Tischler, Cover array string reconstruction, in: CPM, 2010, pp. 251–259.
[10] J.P. Duval, A. Lefebvre, Words over an ordered alphabet and suffix permutations, ITA 36 (2002) 249–259.
[11] J.P. Duval, T. Lecroq, A. Lefebvre, Border array on bounded alphabet, J. Autom. Lang. Comb. 10 (2005) 51–60.
[12] M.J. Fischer, M.S. Paterson, String-matching and other products, Technical Report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.
[13] F. Franěk, W. Lu, P.J. Ryan, W.F. Smyth, Y. Sun, L. Yang, Verifying a border array in linear time, J. Combin. Math. Combin. Comput. 42 (2002) 223–236.
[14] J. Holub, W.F. Smyth, Algorithms on indeterminate strings, in: Proceedings of the 14th Australian Workshop on Combinatorial Algorithms, 2003, pp. 36–45.
[15] J. Holub, W.F. Smyth, S. Wang, Fast pattern-matching on indeterminate strings, J. Discrete Algorithms 6 (2008) 37–50.
[16] T. I, S. Inenaga, H. Bannai, M. Takeda, Counting parameterized border arrays for a binary alphabet, in: LATA, 2009, pp. 422–433.
[17] T. I, S. Inenaga, H. Bannai, M. Takeda, Verifying a parameterized border array in $O(n^{1.5})$ time, in: CPM, 2010, pp. 238–250.
[18] C.S. Iliopoulos, M. Mohamed, L. Mouchard, K.G. Perdikuri, W.F. Smyth, A.K. Tsakalidis, String regularities with don't cares, Nordic J. Comput. 10 (2003) 40–51.
[19] C.S. Iliopoulos, L. Mouchard, M.S. Rahman, A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching, Math. Comput. Sci. 1 (2008) 557–569.
[20] C.S. Iliopoulos, M.S. Rahman, M. Vorácek, L. Vagner, The constrained longest common subsequence problem for degenerate strings, in: CIAA, 2007, pp. 309–311.
[21] W.F. Smyth, S. Wang, New perspectives on the prefix array, in: Proceedings of the 15th SPIRE, in: LNCS, vol. 5280, Springer, 2008, pp. 133–143.
[22] W.F. Smyth, S. Wang, An adaptive hybrid pattern-matching algorithm on indeterminate strings, Internat. J. Found. Comput. Sci. 20 (2009) 985–1004.